

## **Tipos genéricos:**

Um tipo genérico é um tipo que recebe como argumento outros tipos.

## **Exemplo de uma classe genérica:**

```
public class Ponto<T> {  
    private T x;  
    private T y;  
  
    public Ponto(T x , T y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

## Tipos genéricos:

```
public T getX() { return x; }
```

```
public void setX(T x) { this.x = x; }
```

```
public T getY() { return y; }
```

```
public void setY(T y) { this.y = y; }
```

**@Override**

```
public String toString() {  
    return "Ponto{" + "x=" + x + ", y=" + y + '}';  
}
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

- Para criar um objecto do tipo Ponto é necessário dizer ao compilador que tipo vai substituir T, criando um tipo parametrizado

Exemplos:

```
public static void main(String[] args) {
```

```
    Ponto<Integer> i;
```

```
    Ponto<Double> d;
```

```
    i = new Ponto<Integer>(2 , 3);
```

```
    // autoboxing - conversão automática de int para Integer
```

```
    // a conversão de Integer para int designa-se por unboxing
```

```
    d = new Ponto<Double>(4.0 , 5.0);
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
System.out.println("i= " + i);  
System.out.println("d= " + d);
```

Output:

```
i= Ponto{x=2, y=3}  
d= Ponto{x=4.0, y=5.0}
```

-----

Podemos simplificar:

```
i = new Ponto<>(2 , 3);  
d = new Ponto<>(4.0 , 5.0);
```

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Permite a construção de:

- Tipos de dados genéricos (**classes e interfaces**);
- **Métodos** genéricos;

Exemplos de **classe** genérica

Declaração: `public class ArrayList<E> ... //Parâmetro E genérico`

Uso:

`ArrayList<Ponto> listaPontos; //Parâmetro Ponto concreto`

`ArrayList<Pessoa> listaPessoas; //Parâmetro Pessoa concreto`

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
ArrayList x = new ArrayList();
```

```
x.add ("Olá");
```

```
x.add("Adeus");
```

**String y = x.get(0);** Não compila pois devolve um valor do tipo Object

```
String y = (String)x.get(0);
```



```
ArrayList<String> x = new ArrayList<>();
```

```
x.add ("Olá");
```

```
x.add("Adeus");
```

```
String Y = x.get(0) --- Funciona
```

Tipos parametrizados permitem verificação de tipos.

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## Exemplos **de interfaces** genéricas

Declaração: `public interface Map<K, V> ...`

Uso: `Map<Integer, String> m;`

`Map<Jogador, Equipa> futebol;`

## Exemplo de **métodos** genéricos:

`public static void shuffle( List<?> list)`

? Significa: qualquer tipo

Uso: `Collections.shuffle(listaPontos)`

`Collections.shuffle(listaPessoas)`

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## Exemplos de interface genérica

Declaração: `public interface Map<K, V> ...`

```
public interface List<E> ... {  
    ...  
    boolean add(E e);  
}
```

Uso: `Map<Integer, String> pt;`  
`ArrayList<Pessoa> pe;`

```
public interface List<String> ... {  
    ...  
    boolean add(String e);  
}
```



# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Tipos parametrizados podem ser usados em:

- variáveis locais ou variáveis de instancia (atributos);
- parâmetros de métodos ou construtores;
- tipos de retorno (de métodos)

```
public class Exemplo{  
    private List<String> variavel;  
    ...  
    public Exemplo (List<String> lista){ ... }  
  
    public void metodo1(List<Pessoa> lista){ ... }  
  
    public List<String> metodo2(){  
        List<String> lista = new ArrayList(); ...    return lista;  
    }  
}
```

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Convenção Java:

**Nome de Variável**

**Tipo Genérico**

**Significado**

E Tipo de elemento de uma coleção

K Tipo key num mapa

V Tipo value num mapa

T Tipo genérico

S, U Tipos genéricos adicionais

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Numa classe não genérica podemos definir métodos de classe genéricos:

```
public class Serviços {  
    public static <T> void imprimir( T[] vetor ) {  
        for (int i =0; i<vetor.length; i++)  
            System.out.println (vetor[i]);  
    }}
```

Apenas indica que o método, tem um tipo genérico T

```
public class teste {  
    public static void main(String[] args) {  
        String [] lista= {"AAAA", "BBB", "CCCCC", "DDDD"};  
        Serviços.imprimir(lista);  
    }  
}
```

## Genéricos e Herança

- Herança em parâmetros de tipo concreto não implica herança em classes parametrizadas.

Exemplo:

`ArrayList<Figura>` e `ArrayList<Retangulo>`

Retângulo ser subclasse de Figura, não implica que `ArrayList<Retangulo>` seja subclasse de `ArrayList<Figura>`

Não é possível ter:

```
ArrayList<Retangulo> listaRetangulos = new ArrayList<>();
```

```
ArrayList<Figuras> listaFiguras = listaRetangulos; ERRADO
```

## Genéricos e Herança

Para restringir os tipos concretos que podem ser passados a parâmetro genéricos, são usados *Wildcards*:

Wildcard com restrição superior: **< ? extends B >**

Qualquer classe que seja do tipo B ou de um seu subtipo.

Wildcard com restrição inferior: **< ? super B >**

Qualquer classe que seja do tipo B ou de um seu supertipo.

## Genéricos e Herança

Wildcard sem restrição: `< ? >` - **Qualquer tipo**

Exemplos:

```
public class ArrayList<E> ... {  
    ...  
    public ArrayList(Collection<? extends E> c) { ... }  
}
```

...

*Programação Orientada a Objectos - P. Prata, P. Fazendeiro*  
**Genéricos e Herança**

Exemplos:

```
public class ArrayList<E> ... {  
...  
    public void sort(Comparator<? super E> c){ ... }  
}
```

```
public class ArrayList<E> ... {  
...  
    public boolean removeAll(Collection<?> c)  
}
```

## **Ordenação**

Classes que envolvem ordenação implementam uma de duas interfaces:

Interfaces **Comparable** e **Comparator**.

```
import java.util.Comparator;
```

```
import java.lang.Comparable;
```

- São interfaces genéricas e descrevem comparações entre objectos.



## Interface **Comparable**

```
public interface Comparable <T> {  
    public int compareTo ( T outro );  
}
```

Valor retornado pela função compareTo:

**< 0** se o objecto recetor é menor que objeto recebido como parâmetro;

**= 0** se o objecto recetor e o objeto recebido como parâmetro são iguais usando o método (**equals**);

**> 0** caso contrário

**Exemplo:**

```
public class Conta implements Comparable<Conta> {
private long numConta; // número da conta
private String nome; // nome do titular
private double saldo; // saldo actual

public Conta (int n) { numConta = n; nome = "", saldo = 0.0;}
//...
public String toString() {
    return "Conta{" + "numConta=" + numConta + ", nome=" + nome +
", saldo=" + saldo + '}';
}
```

A interface Comparable é implementada por classes que necessitam de ter uma ordem nos seus elementos.



## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

// considerando que duas contas **são iguais se tiverem o mesmo número** de conta:

```
public boolean equals(Object obj) {  
    if ( obj!= null && this.getClass() == obj.getClass()) {  
        return numConta==((Conta)obj).numConta;  
    }  
    return false; }  

```

```
public int compareTo(Conta outro) {  
  
    if (this.numConta > outro.numConta) return 1;  
    else if (this.equals(outro)) return 0;  
    else return -1;  
} ...
```

**Implementam instâncias da interface Comparable as classes:**

**String**            implementa Comparable<String>

**Byte**             implementa Comparable<Byte>

**Integer**          implementa Comparable<Integer>

**Double**

... // todas as classes que “embrulham” os tipos primitivos

...

Se tivermos uma coleção de objetos que pertencem a classes que implementam a interface Comparable, podemos ordenar os seus elementos com o método **sort** da classe Collections.

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Exemplos:**

```
String s1 = "XPTO";
```

```
String s2 = "ABC";
```

```
System.out.println( s1.compareTo(s2) );
```

```
System.out.println( s2.compareTo(s1) );
```

```
System.out.println( s2.compareTo(s2) );
```

**Output:**

**23**

**-23**

**0**

## Exemplos:

```
ArrayList<String> lista;  
lista = new ArrayList<String>();
```

```
lista.add("Maria"); lista.add("José"); lista.add("Costa");  
lista.add("Zeferino"); lista.add("António");
```

```
System.out.println(lista);  
Collections.sort(lista);  
System.out.println(lista);
```

**Output**

```
[Maria, José, Costa, Zeferino, António]  
[António, Costa, José, Maria, Zeferino]
```

**Se quisermos uma comparação de objetos que “não use” o método equals, podemos usar a interface Comparator:**

## Interface **Comparator**

```
public interface Comparator <T> {  
    public int compare ( T outro1, T outro2 );  
}
```

Valor retornado pela função compare:

**< 0** se o objecto outro1 é menor que outro2;

**= 0** se outro1 é “igual” a outro2;

**> 0** caso contrário

**Supondo que queremos ordenar os objetos do tipo Conta por nome de titular. Definimos uma classe que implementa a interface `Comparator<Conta>`:**

```
import java.util.Comparator;
```

```
public class ComparaNomeConta implements Comparator<Conta> {
```

```
    public int compare (Conta c1, Conta c2){  
        return c1.getNome().compareTo (c2.getNome());  
    }  
}
```



## **Collections.sort ( List<T> list )**

```
static <T extends Comparable<? super T>> void sort (List<T> list)
```

- Ordena a lista por ordem ascendente de acordo com a **ordem natural** dos seus elementos.

## **Collections.sort (List l , Comparator c )**

```
static <T> void sort (List<T> list, Comparator<? super T> c)
```

- Ordena a lista de acordo com a ordem estabelecida pelo **Comparator** dado.

**Classe teste:**

**Conta x1 = new Conta (3);**

**Conta x2 = new Conta (2);**

**Conta x3 = new Conta (1);**

**x1.setNome("Maria");**

**x2.setNome("Ana");**

**x3.setNome("Beatriz");**

**ArrayList<Conta> lista2 = new ArrayList<>();**

**lista2.add(x1);**

**lista2.add(x2);**

**lista2.add(x3)**

**Classe teste:**

```
System.out.println(lista2);
```

```
// ordenar por número de Conta
```

```
Collections.sort(lista2);
```

```
System.out.println(lista2);
```

```
// ordenar por nome
```

```
Collections.sort(lista2, new ComparaNomeConta());
```

```
System.out.println(lista2);
```

```
[Conta{numConta=3, nome=Maria, saldo=0.0}, Conta{numConta=2, nome=Ana, saldo=0.0},  
Conta{numConta=1, nome=Beatriz, saldo=0.0}]
```

```
[Conta{numConta=1, nome=Beatriz, saldo=0.0}, Conta{numConta=2, nome=Ana, saldo=0.0},  
Conta{numConta=3, nome=Maria, saldo=0.0}]
```

```
[Conta{numConta=2, nome=Ana, saldo=0.0}, Conta{numConta=1, nome=Beatriz, saldo=0.0},  
Conta{numConta=3, nome=Maria, saldo=0.0}]
```

## **Java Collections Framework (JCF)**

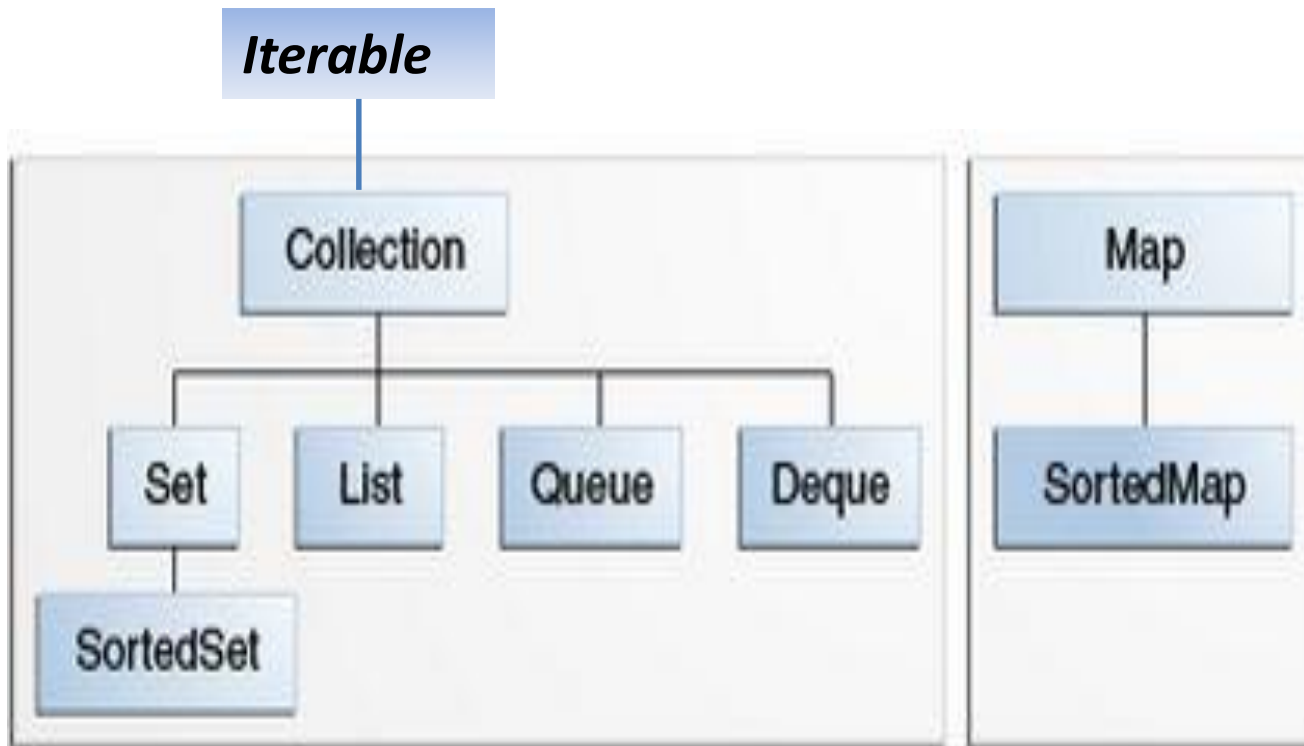
Uma coleção (collection) é um objeto que agrupa vários elementos.

A JCF consiste numa arquitetura para representação e manipulação de coleções. Contém:

- **Um conjunto de Interfaces**
- **Um conjunto de Implementações**
- **Um conjunto de algoritmos**  
(Ex.lo: pesquisa, ordenação, ... )

## Java Collections Framework (JCF)

### Interfaces:



**Duas hierarquias distintas.**

## **Interfaces:**

**Todas as interfaces anteriores são genéricas**, isto é, são declaradas como:

```
public interface Collection<E>...
```

Quando se declara uma instância de uma coleção deve-se especificar o tipo de objetos que a coleção contém.

## **Interfaces:**

**Collection** — Interface raiz da hierarquia de coleções

**Set** — coleção que **não** pode conter elementos duplicados

**List** — coleção **ordenada** de elementos.

- Listas podem conter elementos duplicados;
- Os elementos de uma lista podem ser acedidos através da sua posição (um índice do tipo inteiro) ;
- Um objeto do tipo **ArrayList** é uma Lista.

## **Interfaces:**

**Queue** — coleção usada para a guardar elementos antes de serem processados.

- Tipicamente ordenam os elementos segundo uma ordem (**FIFO** (first-in, first-out))
- **Priority** queues, podem ordenar os elementos de acordo com uma ordem dada pelo utilizador.
- O elemento do topo da queue é sempre o primeiro a ser removido ;
- Numa queue (fila) FIFO todos os elementos são inseridos no final da fila.



## **Interfaces:**

### **Deque — (double ended queue)**

Coleções que podem ser usadas como FIFO (first-in, first-out) e como LIFO (last-in, first-out);

- Os novos elementos podem ser inseridos, consultados e removidos em ambas as extremidades;

## **Interfaces:**

**Map** — Correspondências ou (Maps) são coleções de objetos, parametrizadas por dois tipos. `Map<key, Value>`

Um Map é um objeto que faz corresponder (mapeia) chaves com valores;

Um Map não pode conter chaves duplicadas;

Uma chave (key) corresponde no máximo a um valor;

Uma Hashtable é um Map.

## **Interfaces:**

### **SortedSet** — versão ordenada de Set

Um sortedSet é um Set que mantém os seus elementos em ordem ascendente.

### **SortedMap** — versão ordenada de Map

Um SortedMap mantém as suas correspondências em ordem ascendente dos valores de chave.

## **A Interface Collection:**

Contém métodos que executam operações básicas como:

`int size()` - número de elementos da coleção;

`boolean isEmpty()` - verifica se a coleção está vazia,

`boolean contains(Object element)`

- verifica se um elemento pertence à coleção.

`boolean add(E element)` - adiciona um elemento

`boolean remove(Object element)` - remove um elemento

`Iterator<E> iterator()` **\*\***

## **Percorrer uma coleção:**

### **1 . O construtor for-each**

```
for (Object o : collection)
    System.out.println(o);
```

//Escreve cada elemento da coleção/array na consola.

```
Ex. int [] myArray = {10, 20, 30 , 40, 50};
```

```
    for (int x: myArray)
        System.out.println( x);
```

## **Percorrer uma coleção:**

```
Ex. ArrayList<String> disciplinas = new ArrayList<String>();  
    disciplinas.add("POO");  
    disciplinas.add("BD");  
    disciplinas.add("TC");  
    disciplinas.add("ED");  
    disciplinas.add("PE");  
  
    for (String s : disciplinas)  
        System.out.println(s);
```

## Percorrer uma coleção:

```
for (String s : disciplinas)
    if (s.equals("POO") )
        disciplinas.remove(s);
System.out.println (disciplinas);
```

O que acontece, se executarmos?

```
Exception in thread "main" java.util.ConcurrentModificationException
  at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
  at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
  at Teste.main(Teste.java:42)
```

## **Percorrer uma coleção:**

E assim???

```
for ( int j = 0 ; j<disciplinas.size(); j++)  
    if (disciplinas.get(j).equals("POO") )  
        disciplinas.remove(j);
```

```
System.out.println ( disciplinas );
```

O que acontece?

- E se atualizarmos um elemento da lista?



## **Percorrer uma coleção:**

### **2. Iterators**

Um Iterator é um objeto que permite percorrer os elementos de uma coleção

Interface Iterator:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

**Percorrer uma coleção:**

## **2. Iterators**

boolean hasNext() - verifica se a iteração tem mais elementos;

E next() - devolve o próximo elemento da iteração

Void remove() - remove o último elemento devolvido pela operação next()

**Percorrer uma coleção:**

## **2. Iterators**

O método **iterator** da interface `Collection` devolve um objeto do tipo `Iterator`:

```
Iterator<E> iterator()
```

## **Percorrer uma coleção:**

### **2. Iterators**

Ex.los

```
for (Iterator<String> it = disciplinas.iterator(); it.hasNext();)
    System.out.println( it.next());
}
```

```
Iterator<String> it = disciplinas.iterator();
while (it.hasNext()) {
    System.out.println ( it.next());
}
```

## **Implementações**

**(Classes que implementam as interfaces anteriores)**

**Implementações de uso geral (general purpose) mais usadas:**

HashSet,            implementa a interface Set

ArrayList,        implementa a interface List

HashMap,         implementa a interface Map

LinkedList,      implementa a interface Queue

ArrayDeque,     implementa a interface Deque

## **Algoritmos:**

A classe **Collections** contém algumas funções que operam em coleções:

### **1 - Sorting**

. void **sort** ( List<T> list) \*

\*Os elementos da lista têm de implementar a interface Comparable

. void **sort** (List<T> list, Comparator <? super T> c )

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Ex.lo**

```
Collections.sort(disciplinas);
```

```
for (Iterator<String> it = disciplinas.iterator(); it.hasNext();  
     System.out.println( it.next()));
```

```
// ou
```

```
for (String s : disciplinas)  
    System.out.println(s);
```

```
Output:      BD  
            ED  
            PE  
            POO  
            TC
```

## **2 - Shuffling**

**(Baralha os elementos usando um gerador de valores aleatórios)**

. void **shuffle** (List<?> list)

. void **shuffle** (List<?>List , Random rnd)



# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Ex.lo**

```
System.out.println ("Shuffling");  
Collections.shuffle(disciplinas);  
for (String s : disciplinas)  
    System.out.println(s);
```

Um output:

Shuffling

PE

ED

BD

TC

POO

### 3- Manipulação de dados

. **Reverse** (inverte a ordem dos elementos)

**reverse (List<?> list);**

Ex.lo

`Collections.reverse(disciplinas);`

. **Fill** (substitui cada elemento por um valor dado)

**fill (List<? super T> list, T obj)**

Ex.lo

`Collections.fill(disciplinas, "ops");`

### **3- Manipulação de dados**

- . **copy** - copia os valores de uma lista para outra
- . **swap** - troca dois elementos dadas as suas posições
- . **addAll** – adiciona um conjunto de elementos a uma lista

## 4 - Pesquisa

. binarySearch

Pesquisa um valor (key) numa lista **ordenada**.

```
Collections.sort(disciplinas);
```

```
int pos = Collections.binarySearch(disciplinas, "XPTO");
```

Se a lista contém o valor, devolve a sua posição, caso contrário, devolve o valor **pos** tal que  **$-(pos) - 1$**  é o ponto onde deve ser inserido o elemento na lista, isto é; o índice do primeiro elemento maior do que o valor pesquisado ou o list.size() se todos os valores da lista forem menores que o valor pesquisado.

## 4 - Pesquisa

```
ArrayList<String> lista2 = new ArrayList<>();  
lista2.add("XXX");  
lista2.add("YYYY");  
lista2.add("ZZZZZ");
```

```
Collections.sort(lista2);
```

```
int pos1 = Collections.binarySearch(lista2, "XPTO");
```

```
System.out.println (pos1); pos1 -> -1 inserir em 0
```

```
int pos2 = Collections.binarySearch(lista2, "YYAB");
```

```
System.out.println (pos2); pos1 -> -2 inserir em 1
```

## **4 - Pesquisa**

```
int pos = Collections.binarySearch(list, key);  
    if (pos < 0)  
        list.add(-pos-1, key);
```

- Pesquisa o valor (key) na lista e, caso não esteja na lista, insere.

## **5 – Máximo e Mínimo**

**String s**

**s = Collections.max(disciplinas);**

**s = Collections.min(disciplinas);**